MP 3 – A Unification-Based Type Inferencer

CS 440 – Fall 2006
Revision 1.0

Assigned October 11, 2006
Due October 30, 2006
Extension November 1, 2006

1 Change Log

1.0 Initial Release.

2 Objectives

Your objectives are:

- Become comfortable using record types and disjoint types.
- Understand the unification algorithm.
- Become comfortable with the notation for semantic specifications.
- Understand the type-inference algorithm.

3 Background

One of the major objectives of this course is to provide you with the skills necessary to implement a language. There are three major components to a language implementation: the parser, the internal representation, and the evaluator. In this MP you will work on the middle piece, the internal representation.

An interpreter or a compiler represents an expression in a language with an *Abstract Syntax Tree* (AST), usually implemented by means of a user-defined type. Functions can be written that use this type to perform evaluations, preprocessing... anything that can or should be done with a language. In this MP, you will write some functions that perform type inferencing using unification. This type-inferencer will appear again as a component in several future MPs.

3.1 Type Inferencing

The pattern for type inferencing will be similar to the procedure you used in class to verify a type. The catch is that you are not told the type ahead of time, you have to figure it out as you go. The procedure is as follows:

1. Infer the types of all the subexpressions. For each subexpression, you will get back a proof tree and a list of constraints.
2. Create a new proof tree from the subexpressions.
3. Create a new set of constraints by taking the union of the constraints of the subexpressions. Add any new constraints to this.
4. Return the new proof tree and new set of constraints.

The rules used for a type-inferencer are very similar to the ones used in class. They have one extra component, a field for the constraints. Here's an example:

$$\frac{\Gamma \vdash e_1 \ : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 \ : \tau_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \texttt{int} \mid \{\tau_1 = \texttt{int}\,; \tau_2 = \texttt{int}\} \cup C_1 \cup C_2}$$

The "|" is just some notation to separate the constraint from the expression. This rule says that if you add two expressions e_1 and e_2, which have type τ_1 and τ_2, then the type of the entire expression is integer. But in order for this to work, we constrain the types τ_1 and τ_2 to be of type integer. The final constraint is the union of the constraints inferred from the subexpressions, along with the new constraints for the addition rule.

For example, suppose you want to infer the type of `fun x -> x + 2`. In English, the reasoning would go like this.

1. Let $\Gamma = \{\}$.
2. Examine `fun x -> x + 2`. We don't know what `x` will be, so we let it have type `'1`. Add that to Γ and infer the type of the body...
   (a) Examine `x + 2`. We need to infer the subtypes.
      i. Examine `x`. Γ says that `x` has type `'1`. We do not need to add constraints here.
      ii. Examine `2`. This is an integer. We don't need to add constraints here.
   (b) We combine these inferences together to make a new proof-tree, and say that the result type is `int`. Also, we need to constrain `'1` to be type `int`, and `int` to be type `int`. (Yes, that last one was trivial, but the rule says we have to do it. It will be removed later.)
3. Now we're ready to set the type of the whole expression. The variable `x` has type `'1`, and the output has type `int`, so the whole expression has type `'1 -> int`.
4. Our constraints say to rewrite `'1` as `int` everywhere. We do that, and get a final type of `int -> int`.

Here's a sample run from the MP for the same example.

```
# niceInfer Gamma.empty (FunExp("x",ArithExp(PlusOp,IntExp 2,VarExp "x")));;
{} |- fun x -> 2 + x : ('1 -> int)
  {x:'1; } |- 2 + x : int
    {x:'1; } |- 2 : int
    {x:'1; } |- x : '1
Constraints: [int --> int; '1 --> int; ]
Unifying...New Constraints: ['1 --> int; ]
Substituting...
{} |- fun x -> 2 + x : (int -> int)
  {x:int; } |- 2 + x : int
    {x:int; } |- 2 : int
    {x:int; } |- x : int
- : unit = ()
```

4 Given Code

You are given some initial code in a file `expressions.ml`. Do not modify this file. We may use our own version for grading, or we may need to change some of the implementation.

4.1 Types

Expressions

As mentioned above, an interpreter represents an expression using an AST. We define a type `exp` to represent our language. Not all the elements of the final language are here yet, just enough to give you a feel for the joys of type-checking. Most of the constructors should be self-explanatory. The constructors that take `string` arguments (`FunExp`, `LetInExp`, `LetRecInExp`, and `VarExp`) use the string to represent variable names.

There is a simultaneously defined type `op` that has two constructors, one for integer arithmetic, and one for equality tests. The real interpreter will have a lot more of these, but for type checking this is sufficient.

There are companion functions `showOp` and `showExp` that convert these types into string representations.

Types

In order to express the type of an expression we need a type definition to represent types. Again, these should all be familiar. The `PolyType` constructor takes an integer and represents the `'a` style polymorphic types. The constructor `BotType` represents a type called "bottom" (written $\perp$) and can represent a type error or an unknown type, depending on your mood.

Again, there is a display function for this type.

String Maps

We need an environment to store the types of variables. We will implement it using a `Map` functor. See http://caml.inria.fr/ocaml/htmlman/libref/Map.Make.html for more information about the `Map` functor. For now, we will just say that a functor takes a module and defines another module. The name of the environment module is `Gamma`, a map from strings to types.

There is a `showGamma` function to display the contents of a type environment.

Type Judgments

From lecture, you know that a type judgment has the form $\Gamma \vdash e : \tau$. The `judgment` type is a record that has three fields: `gamma`, `exp`, and `result`, representing Γ, e, and τ, respectively.

The function `showJudgment` returns a string representation.

Proof Trees

A proof tree is a list of assumptions followed by a conclusion. The assumptions are themselves proof trees, and the conclusion is a type judgment. To represent this we have a type `tree` which has two fields: `assume`, a `tree list`, and `conclude`, a judgment.

Guess what `showTree` does? Actually, it's more complicated than the other display functions. It shows the conclusion first, and then the assumptions. Furthermore, the assumptions are indented two spaces in, with help from the `spaces` function.

4.2 Tests

The file `test.ml` has a bunch of tests in it. You will find it useful to go over this code. The `doTests` function takes a list of tests and runs them. There are a lot of tests in this file.

5 Problems

We will give you a file `mp3-given.ml` that you should copy to `mp3.ml`. There are a few functions inside it which proved useful to the staff when writing the solution.

Add your functions to the file. To test them, start up an OCaml interactive session, type `#use "expressions.ml";;` to load the expressions file, and then `#use "mp3.ml"` to load your file. Do **NOT** put a `#use "expressions.ml";;` in your `mp3.ml` file. If you really want to load both in one go, make a third file, and put both `#use` statements in that instead.

5.1 Unification

The first thing you need to do to write a unification-based type inferencer is to write a unifier. A unifier takes a list of pairs of types that are supposed to be equal. Functions, integers, lists, etc. will be the terms in this system, and `PolyType`s will represent variables.

You will remember from lecture that the unification algorithm consists of four transformations. These transformations can be expressed in terms of how an action on the first element of the unification problem affects the remaining elements.

Given a unification problem C, consisting of a head (s, t) and tail C', there are five cases to consider.

1. If s is a variable, and s does not occur in t, output (s, t) as part of the solution. Substitute s with t in C'.
2. If t is a variable, and t does not occur in s, output (t, s) as part of the solution. Substitute t with s in C'.
3. If $s =$ `FunType(`s_1, s_2`)` and $t =$ `FunType(`t_1, t_2`)`, then add (s_1, t_1) and (s_2, t_2) to C'. Discard (s, t). You will do similar things for list types and pair types.
4. If s and t are not variables or "functions" as above, and if they are equal, discard the pair.
5. If none of the above cases apply, it is a unification error.

Problems

1. Write a function `contains : int -> ExpType -> bool`. The first argument is the integer component of a `PolyType`. The second is a target expression. The output indicates whether the variable occurs within the target. This function is used in cases 1 and 2, and prevents recursive types.

```
# contains 1 (FunType(PolyType 1, PolyType 1));;
- : bool = true
# contains 1 (FunType(PolyType 2, PolyType 3));;
- : bool = false
```

2. Write a function `substitute : int -> ExpType -> ExpType -> ExpType`. The first argument is the integer component of a `PolyType`, the second is the replacement value. The third argument is the expression in which to perform the substitution.

```
# substitute 1 IntType (FunType(PolyType 1, PolyType 1));;
- : expType = FunType (IntType, IntType)
# substitute 1 IntType (FunType(PolyType 2, PolyType 3));;
- : expType = FunType (PolyType 2, PolyType 3)
```

3. Now you are ready to write the unification function. Here's a sample run, based on the example given during the unification lecture.

```
# unify;;
- : (expType * expType) list -> (expType * expType) list = <fun>
# unify [ PolyType 1, ListType IntType;
          FunType(PolyType 1, PolyType 1), FunType(PolyType 1, PolyType 2)];;
- : (expType * expType) list =
[(PolyType 1, ListType IntType); (PolyType 2, ListType IntType)]
```

5.2 Inferencing

You are now ready to start writing the type inferencer. We give you one rule for free, to help you get started. The integer rule is:

$$\frac{}{\Gamma \vdash n : \texttt{int} \mid \{\}} \text{assuming } n \text{ is an integer.}$$

The source code:

```
let rec infer gamma exp =
   match exp with
   | IntExp _ ->
         { assume = [];
           conclude = { gamma = gamma;
                        exp = exp;
                        result = IntType } },
         []
   | _ -> raise (Failure "Expression not recognized")
```

There are no assumptions, so the `assume` field is left blank. Also, `gamma` and `exp` are copied as-is. Also, the final `[]` indicates that there are no constraints.

Here's an example of using `infer`:

```
# infer Gamma.empty (IntExp 3);;
- : tree * (expType * expType) list =
(assume = [];  conclude = gamma = <abstr>; exp = IntExp 3; result = IntType,
[])
```

There are two functions, `showTree` and `showConstraints` which will print these out in a nicer format. There is also a function `niceInfer` which does the same thing, but prints a report of what's going on:

```
# niceInfer Gamma.empty (IntExp 3);;
{} |- 3 : int
Constraints: []
Unifying...New Constraints: []
Substituting...
{} |- 3 : int
- : unit = ()
```

Problems

4. Implement the rule for booleans.

$$\frac{}{\Gamma \vdash n : \texttt{bool} \mid \{\}} \text{ assuming } n \text{ is a boolean.}$$

```
# niceInfer Gamma.empty (BoolExp true);;
{} |- true : bool
Constraints: []
Unifying...New Constraints: []
Substituting...
{} |- true : bool
- : unit = ()
```

5. The Variable Rule is:

$$\frac{}{\Gamma \vdash x : \tau \mid \{\}} (\text{if } x : \tau \in \Gamma)$$

The `Gamma.mem` and `Gamma.find` functions may prove to be useful to you.

```
# let g1 = Gamma.add "x" IntType Gamma.empty;;
val g1 : expType Gamma.t = <abstr>
# niceInfer g1 (VarExp "x");;
{x:int; } |- x : int
Constraints: []
Unifying...New Constraints: []
Substituting...
{x:int; } |- x : int
- : unit = ()
```

6. Implement functions. The function rule is this:

$$\frac{\Gamma \cup \{x : \tau_1\} \vdash e_1 \ : \ \tau_2 \mid C}{\Gamma \vdash \texttt{fun}\ x \texttt{->} e_1 \ : \ \tau_1 \rightarrow \tau_2 \mid C}$$

Notice that you have to copy the constraint given by the assumption into the conclusion.

You will need to create a new `PolyType` to write this function. The given function `newPolyType : unit -> expType` will do this for you. You can reset the counter with `resetCounter : unit -> unit`.

```
# niceInfer g1 (FunExp("y",VarExp "x"));;
{x:int; } |- fun y -> x : ('1 -> int)
  {x:int; y:'1; } |- x : int
Constraints: []
Unifying...New Constraints: []
Substituting...
{x:int; } |- fun y -> x : ('1 -> int)
  {x:int; y:'1; } |- x : int
- : unit = ()
# niceInfer g1 (FunExp("y",VarExp "y"));;
{x:int; } |- fun y -> y : ('1 -> '1)
  {x:int; y:'1; } |- y : '1
Constraints: []
Unifying...New Constraints: []
Substituting...
{x:int; } |- fun y -> y : ('1 -> '1)
  {x:int; y:'1; } |- y : '1
- : unit = ()
```

7. Implement arithmetic. There are two rules, one for addition and one for equals.

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash e_1 + e_2 \ : \ \texttt{int} \mid \{\tau_1 = \texttt{int}\,; \tau_2 = \texttt{int}\} \cup C_1 \cup C_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash e_1 = e_2 \ : \ \texttt{bool} \mid \{\tau_1 = \texttt{int}\,; \tau_2 = \texttt{int}\} \cup C_1 \cup C_2}$$

Notice that these are the first in which you will add constraints.

```
# niceInfer g1 (ArithExp(PlusOp,VarExp "x",IntExp 3));;
{x:int; } |- x + 3 : int
  {x:int; } |- x : int
  {x:int; } |- 3 : int
Constraints: [int --> int; int --> int; ]
Unifying...New Constraints: []
Substituting...
{x:int; } |- x + 3 : int
  {x:int; } |- x : int
  {x:int; } |- 3 : int
- : unit = ()
# niceInfer g1 (ArithExp(EqOp, VarExp "x",IntExp 3));;
```

At this point, you can do "real" type checking:

```
# niceInfer Gamma.empty (FunExp("x",ArithExp(PlusOp,VarExp "x",VarExp "x")));;
{} |- fun x -> x + x : ('1 -> int)
  {x:'1; } |- x + x : int
    {x:'1; } |- x : '1
    {x:'1; } |- x : '1
Constraints: ['1 --> int; '1 --> int; ]
Unifying...New Constraints: ['1 --> int; ]
Substituting...
{} |- fun x -> x + x : (int -> int)
  {x:int; } |- x + x : int
    {x:int; } |- x : int
    {x:int; } |- x : int
- : unit = ()
```

8. The next rule to implement is `if`. The rule says that the conditional part must be boolean, and that the second and third subexpressions can be any type at all, as long as they are the same.

$$\frac{\Gamma \vdash e_1 \ : \tau_1 | C_1 \quad \Gamma \vdash e_2 \ : \tau_2 | C_2 \quad \Gamma \vdash e_3 \ : \tau_3 | C_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \ : \ \tau_3 \mid \{\tau_1 = \texttt{bool}\,; \tau_2 = \tau_3\} \cup C_1 \cup C_2 \cup C_3}$$

```
# niceInfer g1 (IfExp (ArithExp (EqOp, VarExp "x", IntExp 3),
                       IntExp 2, IntExp 4));;
{x:int; } |- if x = 3 then 2 else 4 : int
  {x:int; } |- x = 3 : bool
```

```
    {x:int; } |- x : int
    {x:int; } |- 3 : int
  {x:int; } |- 2 : int
  {x:int; } |- 4 : int
Constraints: [bool --> bool; int --> int; int --> int; int --> int; ]
Unifying...New Constraints: []
Substituting...
{x:int; } |- if x = 3 then 2 else 4 : int
  {x:int; } |- x = 3 : bool
    {x:int; } |- x : int
    {x:int; } |- 3 : int
  {x:int; } |- 2 : int
  {x:int; } |- 4 : int
- : unit = ()
```

9. Pairs are easy, because you can put anything into them. But lists have to be checked to make sure that all the elements have the same type.

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \mid C_1 \cup C_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_2 \mid C_2}{\Gamma \vdash [e_1; \cdots e_n] : \tau_1 \texttt{ list} \mid \{\tau_1 = \tau_2; \tau_1 = \tau_3; \cdots; \tau_1 = \tau_n\} \cup C_1 \cdots \cup C_n}$$

The rule for empty lists is slightly different, and as left as an exercise.

```
# let x = VarExp "x";;
val x : exp = VarExp "x"
# niceInfer g1 (PairExp(x,x));;
{x:int; } |- (x,x) : (int * int)
  {x:int; } |- x : int
  {x:int; } |- x : int
Constraints: []
Unifying...New Constraints: []
Substituting...
{x:int; } |- (x,x) : (int * int)
  {x:int; } |- x : int
  {x:int; } |- x : int
- : unit = ()
# niceInfer Gamma.empty (ListExp []);;
{} |- [] : '1 list
Constraints: []
Unifying...New Constraints: []
Substituting...
{} |- [] : '1 list
- : unit = ()
# niceInfer g1 (ListExp [IntExp 3; x]);;
{x:int; } |- [3; x; ] : int list
  {x:int; } |- 3 : int
```

```
  {x:int; } |- x : int
Constraints: [int --> int; ]
Unifying...New Constraints: []
Substituting...
{x:int; } |- [3; x; ] : int list
  {x:int; } |- 3 : int
  {x:int; } |- x : int
- : unit = ()
```

10. Implement function application. The rule is

$$\frac{\Gamma \vdash e_1 \,:\, \tau_1 \mid C_1 \quad \Gamma \vdash e_2 \,:\, \tau_2 \mid C_2}{\Gamma \vdash e_1\ e_2 \,:\, \tau_x \mid \{\tau_1 = \tau_2 \to \tau_x\} \cup C_1 \cup C_2}, \text{for some new } \tau_x$$

    You will infer the type of the function and the argument, and then add a constraint that the input of the function needs to be the same type as the argument.

```
# let g2 = Gamma.add "y" BoolType g1;;
val g2 : expType Gamma.t = <abstr>
# let g3 = Gamma.add "f" (FunType(IntType,BoolType)) g2;;
val g3 : expType Gamma.t = <abstr>
# niceInfer g3 (AppExp(f, IntExp 3));;
{f:(int -> bool); x:int; y:bool; } |- f 3 : '1
  {f:(int -> bool); x:int; y:bool; } |- f : (int -> bool)
  {f:(int -> bool); x:int; y:bool; } |- 3 : int
Constraints: [(int -> bool) --> (int -> '1); ]
Unifying...New Constraints: ['1 --> bool; ]
Substituting...
{f:(int -> bool); x:int; y:bool; } |- f 3 : bool
  {f:(int -> bool); x:int; y:bool; } |- f : (int -> bool)
  {f:(int -> bool); x:int; y:bool; } |- 3 : int
- : unit = ()
# niceInfer Gamma.empty (AppExp(FunExp("x", VarExp "x"), IntExp 3));;
{} |- fun x -> x 3 : '2
  {} |- fun x -> x : ('1 -> '1)
    {x:'1; } |- x : '1
  {} |- 3 : int
Constraints: [('1 -> '1) --> (int -> '2); ]
Unifying...New Constraints: ['1 --> int; '2 --> int; ]
Substituting...
{} |- fun x -> x 3 : int
  {} |- fun x -> x : (int -> int)
    {x:int; } |- x : int
  {} |- 3 : int
- : unit = ()
```

11. Finally, implement `let` and `let rec`.

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \mid C_1 \cup C_2} \qquad \frac{\Gamma \cup \{x : \tau_x\} \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \cup \{x : \tau_x\} \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash \texttt{let rec}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \mid \{\tau_x = \tau_1\} \cup C_1 \cup C_2}$$

```
# niceInfer Gamma.empty (LetInExp("x", IntExp 3, VarExp "x"));;
{} |- let x = 3 in x : int
  {} |- 3 : int
  {x:int; } |- x : int
Constraints: []
Unifying...New Constraints: []
Substituting...
{} |- let x = 3 in x : int
  {} |- 3 : int
  {x:int; } |- x : int
- : unit = ()
# niceInfer Gamma.empty (LetRecInExp("x", IntExp 3, VarExp "x"));;
{} |- let rec x = 3 in x : '1
  {x:'1; } |- 3 : int
  {x:'1; } |- x : '1
Constraints: ['1 --> int; ]
Unifying...New Constraints: ['1 --> int; ]
Substituting...
{} |- let rec x = 3 in x : int
  {x:int; } |- 3 : int
  {x:int; } |- x : int
- : unit = ()
```

6 Conclusions

Congratulations! You now have a Hindley-Milner type system, very much like the one used in OCaml. By doing this, we hope you have learned not only about type-checking and unification, but also will be able to understand what happens when you give code to the compiler to type-check, and what the error messages mean. For that matter, one thing to think about is just how difficult it is to write coherent error messages relating to type errors.